Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

PEDESTRIAN NAVIGATION APPLICATION BASED ON OPENSTREETMAP

TÍTOL DEL PFC: Pedestrian Navigation Application based on OpenStreetMap

TITULACIÓ: Enginyeria de Telecomunicació (segon cicle)

AUTOR: Ramon Tomàs Castellà

DIRECTOR: Sergio Machado Sánchez

DATA: 13 Febrero de 2012

Títol: Aplicación de navegación peatonal basado en OpenStreetMap

Autor: Ramon Tomàs Castellà

Director: Sergio Machado Sánchez

Data: 13 Febrero de 2012

Resumen

El proyecto pretende desarrollar una aplicación web para facilitar la navegación peatonal utilizando el sistema de mapas OpenStreetMap OSM.

Este sistema de mapas de código abierto nos da un gran abanico de posibilidades a la hora de desarrollar nuevas aplicaciones ya que sus mapas son totalmente editables y sus servidores no tienen limitaciones de licencias de uso.

El objetivo es la instalación y configuración de nuestro propio servidor OSM el cual nos proporcionará los mapas con toda la información, ya sean carreteras, puntos de interés, parques, líneas de tren, etc. El otro objetivo es implementar y configurar los servicios Nominatim y XAPI los cuales son de gran ayuda cuando necesitemos hacer peticiones de información sobre diferentes puntos del mapa.

Finalmente se ha desarrollado una aplicación de ruteo en el OSM en el cual podemos hacer la petición de la ruta mas corta entre 2 puntos del mapa y se carga la ruta en el mapa de calles. Para mejorar un poco mas el ruteo y aprovechar la información de los transportes públicos que OSM nos ofrece, se tendrá en cuenta el transporte público cuando la distancia sea demasiado grande para un peatón.

Title: Pedestrian Navigation Application based on OpenStreetMap

Author: Ramon Tomàs Castellà

Director: Sergio Machado Sánchez

Date: February, 13th 2012

Overview

This Project is intended to develop a web application in order to make it easier for pedestrians to navigate using a map system named OpenStreetMap OSM.

This Open-Source map system provides us a wide range of possibilities when it comes to develop a new application as its maps are totally editable and servers do not have license limitations.

The Project aim is to install and configure our own OSM server which is going to give us all the map information such as roads, points of interest, parks, train lines and so on. Another goal is to implement and set up Nominatim and XAPI services which are going to help us when having to make requests to be able to get data from some points or areas on our map.

Finally, it has been developed a routing application on OSM map in which we are able to make shortest path queries between 2 points on our map and it is loaded on street map. In order to enhance routing application and take advantage of public transports which OSM serves us, it is going to take into account public transport when distance between these 2 points is too large for a pedestrian.

CONTENT

CHAPTER 1. INTRODUCTION

1.1 Problem Description

Most powerful street-map applications have experienced an important progress lately when it comes to the great amount of services and information that these types of applications are capable of delivering to users.

They offer a remarkable variety of information pinpointed on a well-designed maps or layers which contain vast amounts of information such as public services, markets, parks, roads and even numbered streets.

On other hand, service applications stretches far beyond as it provides a searcher to be able to find a place on the map, a routing application to seek the shortest path between two points and multiple layers among others.

Although all the advantages have been mentioned above, most of them are license limited applications. Users just are able to carry out a limited amount of requests on the map service in order to make sure that there is not an overloaded use by users.

In addition, users are not allowed to create, edit and update any feature or partial map on the server.

Regarding to the routing service, they base its routing on taking only into account the roads, making it difficult to route a pedestrian user that needs to go far away without being able to take a tube or train line.

1.2 Project Goals

An important goal of this project is to try to implement an Open Source web-based map application and all the services and tools, which will enable us to lay a groundwork to create new services and enhance them.

The first target is to set up and configure our own OpenStreetMap server in order to implement and render a map on a website page.

It is also necessary to build a geocoding server to be able to get geographical points from names-addresses and vice versa. In our case, the server chosen is an Open Source service called Nominatim.

A next step is to add a routing functionality to our OpenStreetMap application. The main purpose of our routing service is to draw on the map the shortest path between 2 points taking into account that it is driving a pedestrian user and therefore, the path should not go through highways or motorways.

In addition, when the distance is too long for pedestrians to be able to reach the destination, the routing algorithm must route them through the tube transport until the destination. Moreover, it must be capable of switching tube lines to try to take the best advantage of all the tube lines spread around a city.

1.3 Project structure

This project is formed by the following structure: Chapter 2 Services and Tools, providing an overview of all the tools and services necessary to set up the project. Chapter 3 Services Configuration, giving a How-To manual about installation and configuration of the services. Chapter 4 Routing Service, a routing scenario is explained and implemented. Chapter 5 Application, describes an overall view of the code and functions in our client-server scenario. Chapter 6 Conclusions and future work, describes a summary of conclusions and several points to be carried out in future improvements.

CHAPTER 2. SERVICES AND TOOLS

2.1 OpenStreetMap OSM

OpenStreetMap is a massive editable world map project born in 2004. The idea behind this is to try to make world maps and enhance them by letting thousands of anonymous contributors make and update their own maps to OSM servers.

OSM provides all the data and software available with free software and free data licenses to everybody is interested in. It means that you are able to use, learn, improve upon and share with others an OSM version.

As it provides freely available tools and data on Internet, it allows us to adapt and use a map according to our necessities by, for instance, creating a map of our neighbourhood, business or cycling track.

It also gives us the advantage of creating a local map in our local server, keep it up to date and customize it for your own purposes without having to be connected to Internet.

Maps are created and improved taking advantage of portable GPS devices, aerial photography or other free sources such as local knowledge or government sources. They have greatly increased the speed of its growing and data has been collected more accurately as well. When large datasets are available, a technical team manage its conversion and import the data into the database.

2.1.1 OSM data elements

OSM map data structure [2] is formed by only a few elements. Each element can be tagged by multiple properties in order to provide information of a

specific point, way or area and be able to link between them creating a map structure.

 **Node**

A node is the basic element of OSM data scheme, which its purpose is to indicate a latitude and longitude of this geospatial point.

This basic element can also be used to describe a way in which a group of point form part of but a node can also be seen as a standalone unconnected point which could for example be defining a point such as a train station or hospital.

 **Way**

A way is a group of points that defines a linear feature such as a street or similar. It must be formed by 2 points as a minimum and 2000 as a top.

Ways are usually tagged with uniform characteristics allowing us to define if a way represents a highway, motorway or residential, its speed, name, etc.

Although a way is tagged with its name street for example, it can be split later into shorter ones in case of having different properties for a way such as different directions or lanes.

 **Area**

Areas are simply elements wrapped by several ways representing a closed surface. They are also tagged defining, for example, buildings, forestlands or water areas or they can be linked to other ways or relations.

 **Relation**

A relation is intended to gather nodes, ways and even other relations in order to be able to represent more complex things such as tube or train routes in a city. Taking tube routes as an example, relations allow us to describe the whole train structure by pointing all the nodes (stations), ways (rails) and other relations (other tube lines) which a relation is composed of.

Tag

A tag is a property attached to node, way or relation. These properties are defined in every element as a Key-Value pair which describes the element (railway = station). OpenStreetMap provides a tagging standard in order to ensure that applications and other users are able to interpret such tags [3].

2.2 PostgreSQL

PostgreSQL [4] is a powerful object-relational database available for most of the main platforms. It is released under free and open license and supported by a global community of developers.

It includes built-in support for quick and vast index searches, an efficient way of dealing with multiple user access by giving them an “snapshot” of the database, adding query rules to a table and create relational objects.

2.2.1 PostGIS

PostGIS is an Open Source module for the PostgreSQL database which add functions to support geographic objects to be able to create a object-relational database based on geographic information systems GIS.

2.3 Mapnik

Mapnik [5] is a tool used for rendering maps on a web-based slippy map by collecting OSM data from a database.

It is an open source toolkit, which is written in C++ and includes a powerful anti-aliasing rendering system with subpixel accuracy which allows a clean and smooth geometries. It is available for most of the platforms.

OSM uses Mapnik to render 256x256 pixel tiles, which can be supplied by a tile server that reads the OSM data from a postGIS database.

Instead of reading on-demand from a database, Mapnik provides a pre-renderer function that allows us to create all the tiles previously in order to supply them when they are asked for.

2.4 XAPI

XAPI [6], also known as OSM extended API, is a read-only service that provides an enhanced searching and querying engine based on the standard map request and adds additional ways of querying the data by attaching tag values to the request or bounding boxes.

The service only collects current data from its database as the source database is a mirror of the main OSM database and is updated frequently via diff dumps.

2.4.1 Query map

A query map implements a bounding box to be able to retrieve all the data which is inside this area.

The bounding box are going to get all the nodes spotted in the box, all the ways that at least one node is into the box and any relation that reference them.

The request structure should look like the following one:

```
GET /api/0.6/map?bbox=left,bottom,right,top
```

Where:

- `left` is the longitude of the left (westernmost) side of the bounding box
- `bottom` is the latitude of the bottom (southernmost) side of the bounding box
- `right` is the longitude of the right (easternmost) side of the bounding box
- `top` is the latitude of the top (northernmost) side of the bounding box

Another improving is the large amount of elements the query map is able to achieve, up to 10 million elements per request.

2.4.2 Query Tags

Xapi also defines tag-based queries for node, way and relation elements.

When querying nodes by a tag, the service should return a XML file containing all the nodes related to the tag.

An example is shown below:

```
GET /api/0.6/node[amenity=hospital]
```

Where response:

```
<?xml version='1.0' standalone='no'?>
<osm version='0.6' generator='xapi: OSM Extended API'
 xmlns:xapi='http://www.informationfreeway.org/xapi/0.6'
 xapi:uri='/api/0.6/node[amenity=hospital]'
 xapi:planetDate='200803150826'
 xapi:copyright='2008 OpenStreetMap contributors'
 xapi:instance='zappy2'>
  <node id='672180' lat='48.2111685091189' lon='16.3035366605548' timestamp='2006-09-11T16:28:25+01:00
    <tag  k='amenity' v='hospital'/>
    <tag  k='name' v='Wilhelminenspital'/>
  </node>
  <node id='3596186' lat='53.4633699598014' lon='-2.22667910006381' timestamp='2007-06-21T17:10:58+01:00
    <tag  k='amenity' v='hospital'/>
    <tag  k='name' v='Manchester Royal Infirmary'/>
  </node>
  ...
</osm>
```

When querying ways, the service is going to send back a XML file composed by all the ways related to the tag and all the nodes which the ways are composed of.

```
GET /api/0.6/way[landuse=residential]
```

Where response:

```
<?xml version='1.0' standalone='no'?>
<osm version='0.6' generator='xapi: OSM Extended API'
 xmlns:xapi='http://www.informationfreeway.org/xapi/0.6'
 xapi:uri='/api/0.6/way[landuse=residential]'
 xapi:planetDate='200803150826'
 xapi:copyright='2008 OpenStreetMap contributors'
 xapi:instance='zappy2'>
  <node id='218963' lat='52.5611324692581' lon='-1.79024812573334' timestamp='2006-03-22T16:47:48+00:00
  </node>
  <node id='331193' lat='53.7091237972264' lon='-1.50282510180841' timestamp='2007-03-31T00:09:22+01:00
    <tag  k='highway' v='traffic_signals'/>
    <tag  k='source' v='Yahoo'/>
  </node>
  ...
  <way id='4958218' timestamp='2007-07-25T01:55:35+01:00' version='3' changeset='2211'>
    <nd ref='218963'/>
    <nd ref='331193'/>
    ...
    <tag  k='landuse' v='residential'/>
    <tag  k='source' v='landsat'/>
  </way>
</osm>
```

When querying relations, it returns an XML file with all the ways and nodes referenced by the relation introduced in the tag.

```
GET /api/0.6/relation[name=Fonnereau Way]
```

Where response:

```
<?xml version='1.0' standalone='no'?>
<osm version='0.6' generator='xapi: OSM Extended API'
 xmlns:xapi='http://www.informationfreeway.org/xapi/0.6'
 xapi:uri='/api/0.6/way[landuse=residential]'
 xapi:planetDate='200803150826'
 xapi:copyright='2008 OpenStreetMap contributors'
 xapi:instance='zappy2'>
  <node ...
  <way ...
  <relation id='2670' timestamp='2007-10-25T03:05:34Z'
    <member type='way' ref='3992472' role=''/>
    <member type='way' ref='3992524' role=''/>
    <member type='way' ref='4253050' role=''/>
    <member type='way' ref='4253053' role=''/>
    <member type='way' ref='4266813' role=''/>
    <member type='way' ref='10285106' role=''/>
    <tag k='name' v='Fonnereau Way'/>
    <tag k='network' v='Ipswich footpaths'/>
    <tag k='type' v='route'/>
  </relation>
</osm>
```

XAPI queries also provides a combination of multiple values with the same query:

```
GET /api/0.6/way[highway=motorway|motorway_link|trunk|primary]
```

As well as multiple keys:

```
GET /api/0.6/way[amenity|leisure=golf_course]
```

To avoid getting vast amounts of data, it is advisable to use a tag query with a bounding box as the following example:

```
GET /api/0.6/node[amenity=hospital][bbox=2.1,41.1,2.2,41.2]
```

2.5 Nominatim

Nominatim ***(from latin, "by name")*** [7], is a searching tool to find OSM data by its name and address. It also provides a geocoding reverse function to be able to get addresses or names from its geographical points.

When requesting a name or address, nominatim is capable of handling multiple options and parameters.

The following table shows the most important ones:

Options	Description
format	Specify the output format being called. Must be one of the following, if supplied: • html • json • xml
json_callback	Callback function used to display the results below
q	Query string being searched. Search terms are processed left to right and house numbers, where defined, will be used. Commas are optional but will improve performance by reducing the complexity of the search.
addressdetails	Include a breakdown of addresses into separate elements. • **0** - No, do not display the Address Details. • **1** - Yes, display the Address Details.
limit	Limits the number of returned results to the integer entered.
countrycodes	Limits the search to a specific country or a list of countries. • `countrycodes=`*countrycode,countrycode,countrycode...*
viewbox	Preferred area to find search results. • `viewbox=`*left,top,right,bottom*
osm_type	Specify if the feature is a Node, Way, or Relation in OpenStreetMap. • **N** - Node • **W** - Way • **R** – Relation

Table 2.1: Nominatim request options

A typical Nominatim request should look like the one below:

```
http://open.mapquestapi.com/nominatim/v1/search?format=json&addressde
tails=1&json_callback&q=montjuich
```

Where responses a JSON file:


```
- {
      place_id: "57852357",
      licence: "Data Copyright OpenStreetMap Contributors, Some Rights Reserved. CC-BY-SA 2.0.",
      osm_type: "way",
      osm_id: "62324767",
    - boundingbox: [
          "37.6329536437988",
          "37.6334762573242",
          "-1.00530481338501",
          "-1.00293529033661"
      ],
      lat: "37.6332154770323",
      lon: "-1.00412005590469",
      display_name: "Calle Castillo de Montjuich, Los Dolores, Urbanización Castillitos, Cartagena, Murcia, 30310, España'
      class: "highway",
      type: "residential",
    - address: {
          road: "Calle Castillo de Montjuich",
          residential: "Los Dolores",
          suburb: "Urbanización Castillitos",
          city: "Cartagena",
          county: "Murcia",
          state: "Murcia",
          postcode: "30310",
          country: "España",
          country_code: "es"
      }
  }
```


As mentioned previously, it is capable of sending back an address response from a geocoding point as it is shown in the following example:

```
http://open.mapquestapi.com/nominatim/v1/reverse?format=json&addressd
etails=1&json_callback&lon=1.7&lat=41.3
```

2.6 JOSM editor

JOSM [8] is a java tool for editing OSM maps. The tool enables us to download from the server a partial region of the map to modify and enhance it. Once edited, the map is ready to be uploaded to the server and get updated. The user has to be registered previously on its web page in order to be able to upload changes on the server.

JOSM is capable of creating and drawing nodes, ways or areas on the map and tag them.

The figure below shows the JOSM editor screen:

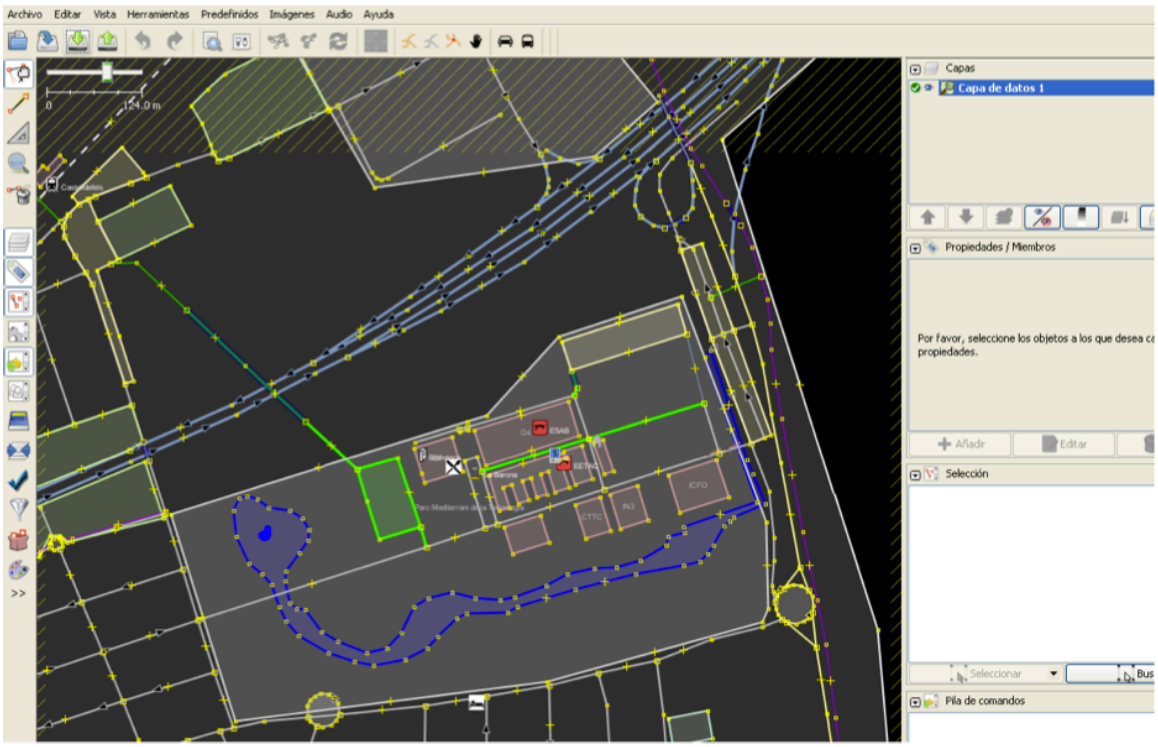


Figure 2.1: JOSM editor screen

JOSM tool has been used in this project to edit station points and tube lines to try to adapt them in order to apply a routing search. Tube lines are isolated from road network as there are no points joining both networks. Therefore, there must be a point between stations and road network to be capable of applying routing through tube lines coming from a road.

CHAPTER 3. SERVICES CONFIGURATION

3.1 OpenStreetMap installation

In this chapter is described the steps on how to install and configure an OpenStreetMap server **[9]** and all the tools and libraries in order to get it work properly. It shall be installed a PostgreSQL database to dump OSM data into it, and OSM2PGSQL tool in order to import the data into database.

Mapnik is necessary too when rendering a map. In addition, there has to be a tile server which serves the maps (tile maps) rendered by Mapnik to our website page.

3.1.1 PostgreSQL database

- Get the latest OSM data file

```
wget http://planet.openstreetmap.org/planet-latest.osm.bz2
```

- Install a PostgreSQL database and its PostGIS extensions

```
apt-get install postgresql-8.4-postgis postgresql-contrib-8.4
apt-get install postgresql-server-dev-8.4
apt-get install build-essential libxml2-dev libtool
apt-get install libgeos-dev libpq-dev libbz2-dev proj
```

- Configure PostGIS

```
gedit /etc/postgresql/8.4/main/postgresql.conf
shared_buffers = 128MB # 16384 for 8.1 and earlier
checkpoint_segments = 20
maintenance_work_mem = 256MB # 256000 for 8.1 and earlier
autovacuum = off
```

- Creating a PostgreSQL database

```
sudo -u postgres -i
createuser username # answer yes for superuser
createdb -E UTF8 -O username gis
createlang plpgsql gis
exit
```

- Set up PostGIS extension on the PostgreSQL database created previously

```
psql -f /usr/share/postgresql/8.4/contrib/postgis-1.5/postgis.sql
-d gis
```

- Setting a username that Mapnik should use to render maps

```
echo "ALTER TABLE geometry_columns OWNER TO username; ALTER TABLE
spatial_ref_sys OWNER TO username;" | psql -d gis
```

- Set the Spatial Reference Identifier on the new database

```
psql -f ~/bin/osm2pgsql/900913.sql -d gis
```

3.1.2 OSM2PGSQL Tool

- Install osm2pgsql tool

```
svn co http://svn.openstreetmap.org/applications/utils/export/
osm2pgsql/
cd osm2pgsql
./autogen.sh
./configure
make
```

- Load the OSM data into the database with osm2pgsql

```
./osm2pgsql -S default.style --slim -d gis -C 2048 --number-
processes=1 --cache-strategy=dense ~/planet/planet-100217.osm.bz2
```

- After this, should have appeared a progress window like this one:

```
Processing: Node(593072k) Way(45376k) Relation(87k)
Exception caught processing way id=110802 Exception caught
processing way id=110803 Processing: Node(593072k)
Way(45376k) Relation(474k)
```

Depending on your hardware, this process could take a long time to finish. In our case, it took 4 weeks to be able to load all the data into de database.

3.1.3 Mapnik

- Install Mapnik dependencies

```
apt-get install libltdl3-dev libpng12-dev libtiff4-dev libicu-dev
apt-get install libboost-python1.40-dev python-cairo-dev python-
nose
apt-get install libboost1.40-dev libboost-filesystem1.40-dev
apt-get install libboost-iostreams1.40-dev libboost-regex1.40-dev
libboost-thread1.40-dev
apt-get install libboost-program-options1.40-dev libboost-
python1.40-dev
apt-get install libfreetype6-dev libcairo2-dev libcairomm-1.0-dev
apt-get install libgeotiff-dev libtiff4 libtiff4-dev libtiffxx0c2
apt-get install libsigc++-dev libsigc++0c2 libsigx-2.0-2 libsigx-
2.0-dev
apt-get install libgdal1-dev python-gdal
apt-get install imagemagick ttf-dejavu
```

- Build Mapnik library

```
svn co http://svn.mapnik.org/tags/release-0.7.1/ mapnik
python scons/scons.py configure INPUT_PLUGINS=all OPTIMIZATION=3
SYSTEM_FONTS=/usr/share/fonts/truetype/
python scons/scons.py
python scons/scons.py install
ldconfig
```

- Install Mapnik tools

```
svn co http://svn.openstreetmap.org/applications/rendering/mapnik
```

- Install prepared world boundary data. Mapnik uses prepared files to generate coastlines and oceans.

```
mkdir world_boundaries
wget http://tile.openstreetmap.org/world_boundaries-spherical.tgz
tar xvzf world_boundaries-spherical.tgz
wget http://tile.openstreetmap.org/processed_p.tar.bz2
tar xvjf processed_p.tar.bz2 -C world_boundaries
wget http://tile.openstreetmap.org/shoreline_300.tar.bz2
tar xjf shoreline_300.tar.bz2 -C world_boundaries
wget http://www.naturalearthdata.com/http//www.naturalearthdata.c
om/download/...
unzip 10m-populated-places.zip -d world_boundaries
wget http://www.naturalearthdata.com/http//www.naturalearthdata.c
om/download/...
unzip 110m-admin-0-boundary-lines.zip -d world_boundaries
```

3.2 Map Tile Server

A Tile Server is a server in which a web-based map application is able to request specific maps. Then, once web-map page needs to get some parts of the map, a tile server renders these on-demand tiles to be able to provide them to map application.

When implementing a Tile Server there might be various ways of setting up a map server with Mapnik. These methods are described below.

3.2.1 Tile-Cache

Mod Tile **[10]** is a tool to implement a tile-cache method and serves tiles to be used in a slippy map. It gives us on-demand rendering and an efficient way of caching, making sure that only a small fraction of overall tiles are kept on disk.

- Install Apache web server and tools

```
aptitude install apache2 apache2-threaded-dev apache2-mpm-prefork
apache2-utils
apt-get install libagg-dev
```

- Get and install mod_tile

```
svn co http://svn.openstreetmap.org/applications/utils/mod_tile cd
make
make install
```

- Configure mod_tile

```
gedit /etc/renderd.conf
plugins_dir=/usr/local/lib/mapnik/input
font_dir=/usr/lib/mapnik/fonts XML=/home/ubuntu/bin/mapnik/osm.xml
HOST=localhost
```

- Configure Apache for mod_tile

```
sudo nano /etc/apache2/conf.d/mod_tile # new file
LoadModule tile_module /usr/lib/apache2/modules/mod_tile.so
```

- Configure default website

```
LoadTileConfigFile /etc/renderd.conf ModTileRenderdSocketName
/tmp/osm-renderd
# Timeout before giving up for a tile to be rendered
ModTileRequestTimeout 0
# Timeout before giving up for a tile to be rendered that is
otherwise missing ModTileMissingRequestTimeout 30
```

- Start renderd daemon

```
cd ~/src/mod_tile
./renderd
```

3.2.2 Pre-Rendered Tiles

This method generates map tiles which are served by an Apache web server. Instead of generating on-demand tiles, it creates all the tiles previously and stores them into a directory. When OpenStreetMap requests a certain map tiles, it just provides the pre-rendered tiles without having to render anymore. That scenario has a clear advantage as it does not need to read from the database and render new tiles but, in terms of memory, it is going to consume much more memory on disk than the tile-cache configuration which requests on-demand map tiles. Therefore, its cost in terms of memory is massive when trying to render wide areas.

To be able to pre-render map tiles the next command has to be applied:

```
MAPNIK_MAP_FILE="osm.xml" MAPNIK_TILE_DIR="tiles/"./generate_tiles.py
```

The command needs to know the osm.xml configuration file and in what directory Mapnik has to store the tiles.

Generate_tiles.py python script needs to be configured too in order to set a bounding box to render a specific area and how many zoom levels you want to achieve.

```
# Spain+
    bbox = (-10.62,35.33, 5.59,44.69)
    render_tiles(bbox, mapfile, tile_dir, 1, 19 , "spain+")
```

In our case, we have decided to use a tile-cache server. Despite the fact that the server do have to render tiles in some requests when there is no cached tiles, its consumption on disk is going to be tremendously lower.

3.2 Nominatim

Despite its guide about how to install a Nominatim server **[11]**, it is been impossible for us to be able to build our own Nominatim service. Nevertheless, in this subchapter is going to be described the steps to be followed in order to build this service.

- Create a database for Nominatim

```
sudo su postgres
createdb gazetteer
createlang plpgsql gazetteer
```

- Add PostGIS extensions to database

```
cat /usr/share/postgresql/8.4/contrib/_int.sql | psql gazetteer
cat /usr/share/postgresql/8.4/contrib/pg_trgm.sql | psql gazetteer
cat /usr/share/postgresql/8.4/contrib/postgis-1.5/postgis.sql | psql
gazetteer
cat /usr/share/postgresql/8.4/contrib/postgis-1.5/spatial_ref_sys.sql
| psql gazetteer
```

- Import OSM data

```
./osm2pgsql -lsc -O gazetteer -C 2000 -d gazetteer planet.osm.bz2
```

- Add Gazetteer functions to database. Gazetteer is a geographical dictionary which contains information about places and place names that are linked to a map.

```
cat gazetteer-tables.sql | psql gazetteer
cat gazetteer-functions.sql | psql gazetteer
```

- Copy Gazzetteer data into database.

```
cat gazetteer-loaddata.sql | psql gazetteer
```

- Index the database

```
cat gazetteer-index.sql | psql gazetteer
```

- Set up a website. Gazzetteer provides a PHP script to be able to implement a web service searcher.

```
cp website/* ~/public_html/
```

CHAPTER 4. ROUTING SERVICE

4.1 Goals to achieve

The aim of this chapter is to implement a routing service on the OpenStreetMap application in order to indicate the shortest path between 2 geographical points introduced by users.

Once built a routing service, a next target is to improve it by adding new functionalities such as tube transport routing and switching. It should lay a groundwork to create or improve new routing services on OpenStreetMap.

4.2 Scenario Description

Our first solution was to try to extract all the ways and points from XAPI service in order to process the data and find the shortest path by applying a Dijkstra shortest path algorithm. When requesting to XAPI server all the data bounded in our path, the routing service should create a network topology in order to be able to apply weights and Dijkstra algorithm.

However, after testing a XAPI service we noticed that XAPI service had too much latency when requesting a small bounding box area. Taking into account that a routing application needs to be as fast as possible, it is a limitation we cannot assume.

Figuring out others ways of implementing a routing service, we came across a project called PgRouting.

PgRouting **[12]** is an Open-source library which adds routing functionalities to PostGIS database. It basically implements a Dijkstra algorithm in a PostGIS

database in such a way that it allows you to prepare the data and make routing queries to a database.

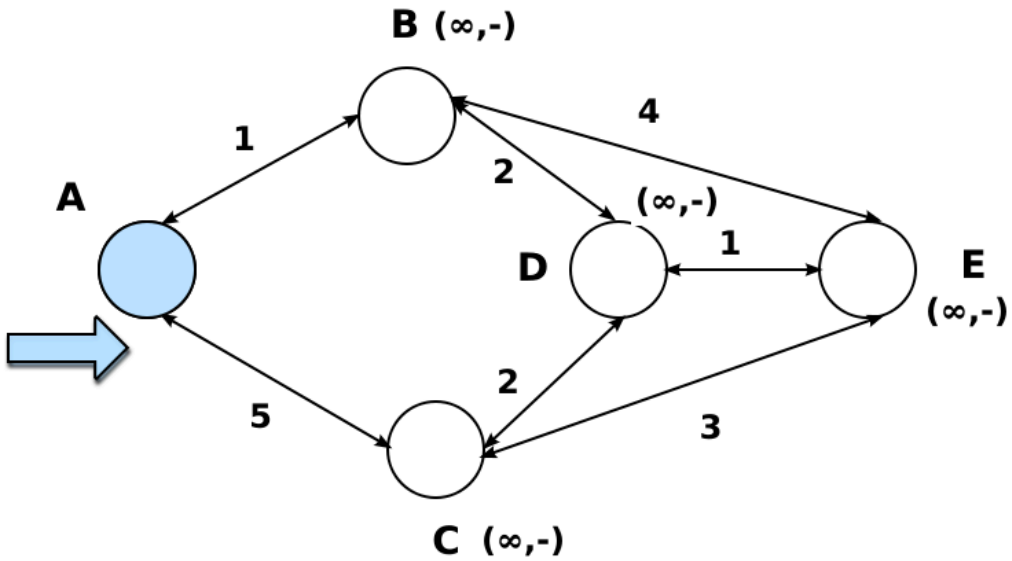
The next chapter is going to explain how to set up and configure PostGIS to add pgRouting functionalities to our database.

4.3 Dijkstra algorithm

Dijkstra is a graph searching algorithm which is conceived to find from a single source, a shortest path tree. This algorithm is often used for routing such as routing protocols in networks.

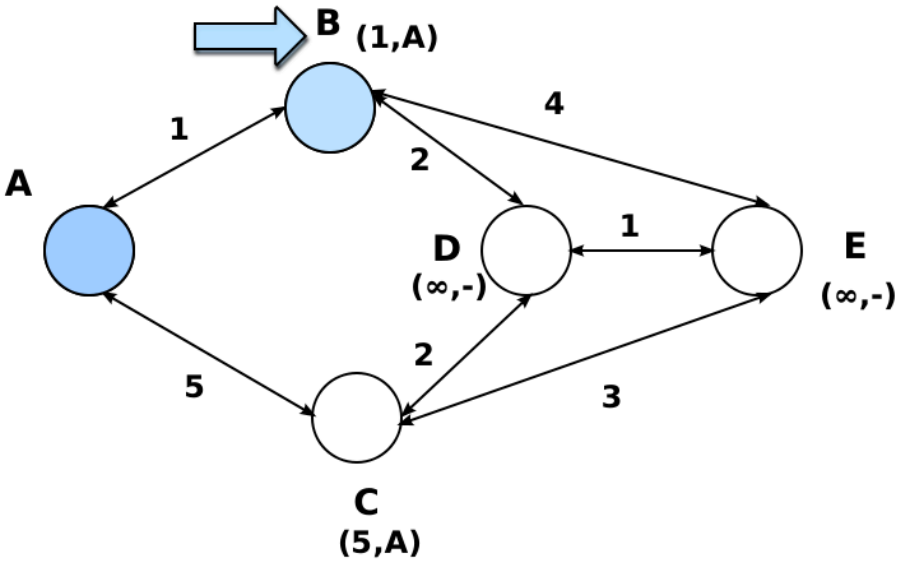
Given a source vertex in a graph, the algorithm is able to track down a path with lowest cost between its source and any other point in the graph.

To start, mark the distance to each vertex on the graph with an infinite value. This is just done to indicate that a vertex has not been visited yet.

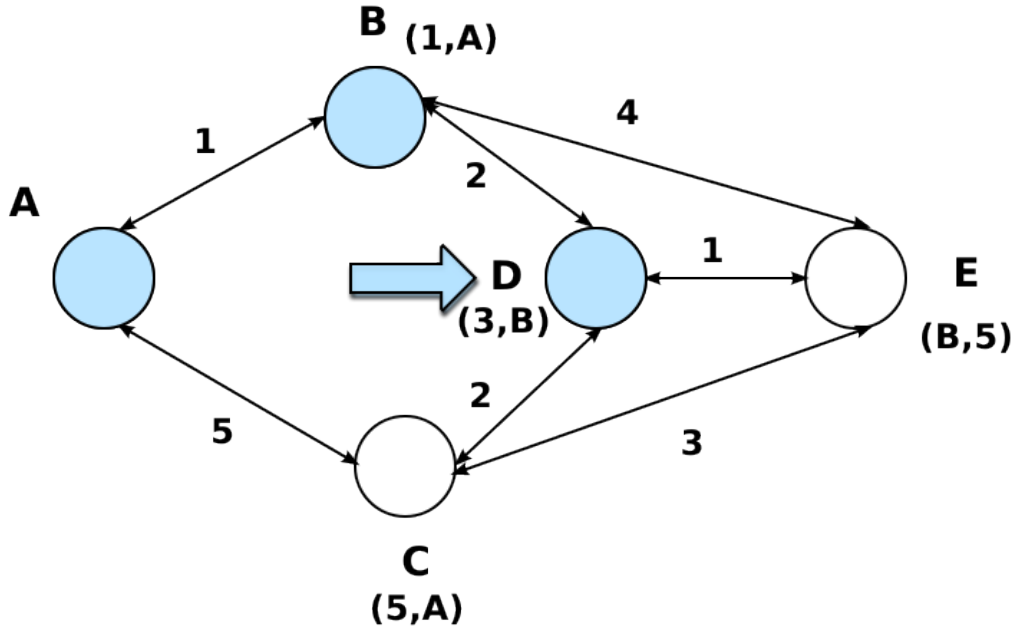


Taking as a reference our source point, in the first iteration we have to update the distance to each neighbour vertex connected to us. This is done by determining the sum of the distances between our current point and the value of the vertex to check.

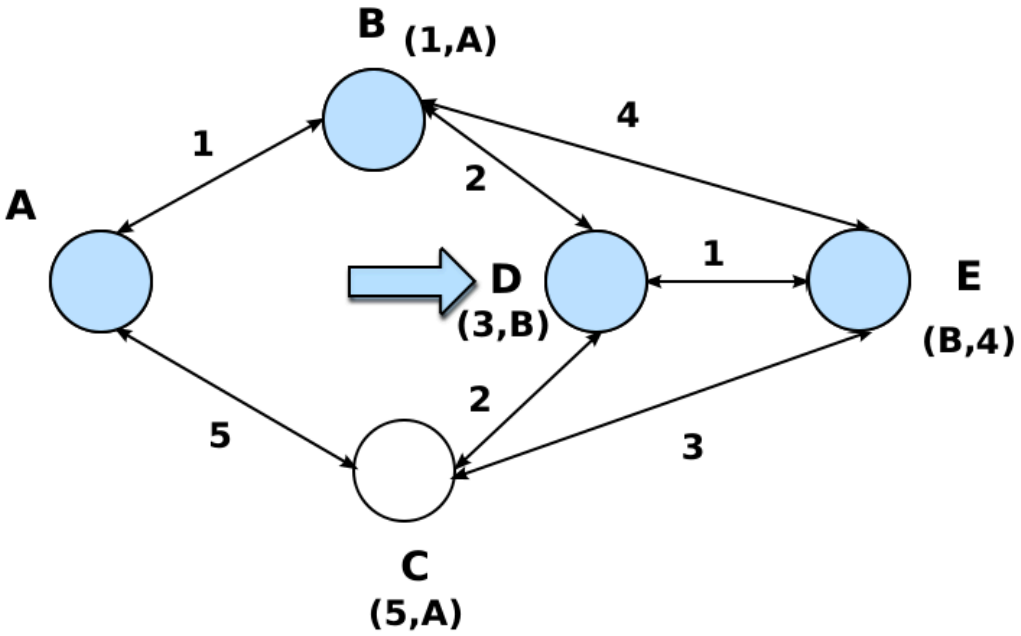
After updating vertex B and C, the minimum distance is taking vertex B as it has a cost of 1.



The next step for vertex B is to check its neighbours to find a shortest path again. After updating vertex D and E, the algorithm notices that the shortest path is vertex D as it only has to make 3 hops to reach this one.



Finally, vertex D checks out its neighbours C and E. As the sum to reach C is 5 and vertex C label has got the same value it is not necessary to update its label. When it comes to vertex E, we notice that to reach this point we just must take 4 hops and then, we are able to update its label as it has a lower cost. Once updated their labels, vertex E will be chosen as it is got a lower cost than vertex C.



The simplest implementation of Dijkstra algorithm stores vertices in an array in order to be checked later. When a vertex checks its neighbours, it searches that array of possible candidates in order to find the shortest one. The general formula to assess its cost in terms of running time is described as $O(|E|.t_{dk} + |V|.t_{em})$, where |V| represents all the vertices to be processed and |E| all their edges taking into account the time to decrease key t_{dk} and extract the minimum t_{em} from array of vertices Q.

Then, it is just a simple search through all the vertices and thus, the cost of extracting de minimum from a simple array of vertices is going to take a running time of $O(|E| + |V|^2)$ as algorithm checks out the entire array of vertices V in order to extract the minimum one and this process must be done for all the vertices of the graph. When it comes to edges E, it is going to take a cost of $O(|E|)$ caused by the fact that algorithm does not need to reorder the queue of vertices when decreasing key.

Dijkstra algorithm can be implemented more efficiently by applying a min-priority queue, which stores the vertices taking into account its priority. It enhances a lot its running time as the queue gives more priority to vertex with

lower costs and therefore, current vertex is going to check out much less vertices to find the shortest one. Its running time is $O((|E|+|V|)\log|V|)$

Pseudocode is described below:

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:             // Initializations
        dist[v] := infinity ;               // Unknown distance function from source to v
        previous[v] := undefined ;          // Previous node in optimal path from source
    end for ;
    dist[source] := 0 ;                     // Distance from source to source
    Q := the set of all nodes in Graph ;    // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                   // The main loop
        u := vertex in Q with smallest distance in dist[] ;
        if dist[u] = infinity:
            break ;                         // all remaining vertices are inaccessible from source
        end if ;
        remove u from Q ;
        for each neighbor v of u:           // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v) ;
            if alt < dist[v]:               // Relax (u,v,a)
                dist[v] := alt ;
                previous[v] := u ;
                decrease-key v in Q;        // Reorder v in the Queue
            end if ;
        end for ;
    end while ;
    return dist[] ;
end Dijkstra.
```

Figure 4.1: Dijkstra algorithm

4.4 Routing Service installation

4.4.1 pgRouting installation

First of all, pgRouting tools shall be installed.

```
# Add pgRouting launchpad repository
add-apt-repository ppa:georepublic/pgrouting
apt-get update
# Install pgRouting packages
apt-get install gaul-devel \
postgresql-8.4-pgrouting \
postgresql-8.4-pgrouting-dd \
postgresql-8.4-pgrouting-tsp
# Install osm2pgrouting package
```

```
apt-get install osm2pgrouting
# Install workshop material (optional)
apt-get install pgrouting-workshop
```

Then, we create a database in PostGIS named routing.

```
# Create database "routing" createdb -U postgres -T template_routing
routing
```

Add pgRouting functions to routing database.

```
# become user "postgres" (or run as user "postgres")
sudo su postgres
# create routing database
createdb routing createlang plpgsql routing
# add PostGIS functions
psql -d routing -f /usr/share/postgresql/8.4/contrib/postgis-
1.5/postgis.sql
psql -d routing -f /usr/share/postgresql/8.4/contrib/postgis-
1.5/spatial_ref_sys.sql
# add pgRouting core functions
psql -d routing -f /usr/share/postlbs/routing_core.sql
psql -d routing -f /usr/share/postlbs/routing_core_wrappers.sql
psql -d routing -f /usr/share/postlbs/routing_topology.sql
```

4.4.2 Loading OSM data

Once created and added routing functions to our database is time to load OSM data into it. pgRouting provides a tool named osm2pgrouting which helps us import OSM data to database.

When importing data with osm2pgrouting, it only takes nodes and ways as long as they are specified in mapconfig.xml file used to import data. Thus,

mapconfig.xml has to specify all the types of ways and nodes you want to include for routing.

An example is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
```

```
    <type  name="highway" id="1">
        <class name="motorway" id="101" />
        <class name="motorway_link" id="102" />
        <class name="motorway_junction" id="103" />
    </type>
    <type name="junction" id="4">
        <class name="roundabout" id="401" />
   </type>
</configuration>
```

Run the converter osm2pgrouting.

```
osm2pgrouting -file "data/sampledata.osm" \
-conf "/usr/share/osm2pgrouting/mapconfig.xml" \
-dbname routing \
-user postgres \
-clean
```

When the converter is done, it creates several tables in which a network topology is created. The database should look like this:

```
                List of relations
Schema |       Name         | Type  |  Owner
-------+--------------------+-------+----------
public | classes            | table | postgres
public | geography_columns  | view  | postgres
public | geometry_columns   | table | postgres
public | spatial_ref_sys    | table | postgres
public | types              | table | postgres
public | ways               | table | postgres
(6 rows)
```

Basically, the table which contains all the network data is named as ways. It is structured as follows:

```
              Table "public.ways"
Column   |       Type        | Modifiers
---------+-------------------+-----------
gid      | integer           | not null
class_id | integer           |
```

```
length    | double precision |
name      | character(200)   |
the_geom | geometry          |
Indexes:     "ways_pkey" PRIMARY KEY, btree (gid)
            "geom_idx" gist (the_geom)
Check constraints:
"enforce_dims_the_geom" CHECK (ndims(the_geom) = 2)
"enforce_geotype_the_geom" CHECK (geometrytype(the_geom) =
'MULTILINESTRING'::text OR the_geom IS NULL)
"enforce_srid_the_geom" CHECK (srid(the_geom) = 4326)
```

Where gid is an internal index to be able to identify road links, class_id gives us the type of street, its length, name of street, and the_geom indicates all the lines a way is made of and a geographical position of each line.

Add a source and target column to ways table in order to assign an ID to both vertices of a way.

```
-- Add "source" and "target" column
ALTER TABLE ways ADD COLUMN "source" integer;
ALTER TABLE ways ADD COLUMN "target" integer;  -- Run topology
function
```

When dealing with a massive amount of data, it is recommendable to add indices in order to speed up our searching.

```
CREATE INDEX source_idx ON ways("source");
CREATE INDEX target_idx ON ways("target");
```

4.4.3 Shortest Path Queries

Shortest Path Queries are sql commands defined in pgRouting to make Dijkstra requests. Queries just require a source and target attribute to be able to return its shortest path.

```
SELECT gid, AsText(the_geom) AS the_geom FROM dijkstra_sp('ways',
5700, 6733);
```

Where its response is a table with all the ways the shortest path is made of:

```
  gid   |                                the_geom
--------+-------------------------------------------------------------
5534 | MULTILINESTRING((-104.9993415 39.7423284, ... ,-104.9999815
39.7444843))
5535 | MULTILINESTRING((-104.9999815 39.7444843, ... ,-105.0001355
39.7457581))
5536 | MULTILINESTRING((-105.0001355 39.7457581,-105.0002133
39.7459024))     ... | ...
19914 | MULTILINESTRING((-104.9981408 39.7320938,-104.9981194
39.7305074)) (37 rows)
```

It is also possible to limit your searching area by adding a bounding box:

```
SELECT gid, AsText(the_geom) AS the_geom FROM dijkstra_sp_delta
('ways', 5700, 6733, 0.1);
```

CHAPTER 5. APPLICATION

In Application chapter is going to be explained the web tools which have helped us build our map-service website as well as server's side.

On other hand, it shall be described a workflow and class diagram for both client and server side to take an overview of application source code and function structure.

5.1 Web-application tools

The main tools used to build up our web-map page are named GeoExt, ExtJS and OpenLayers. They are going to help us to interact, send and get data from servers and present information on map as well. These tools are explained in the following subchapters.

Figure below shows us an overview of the complete structure:

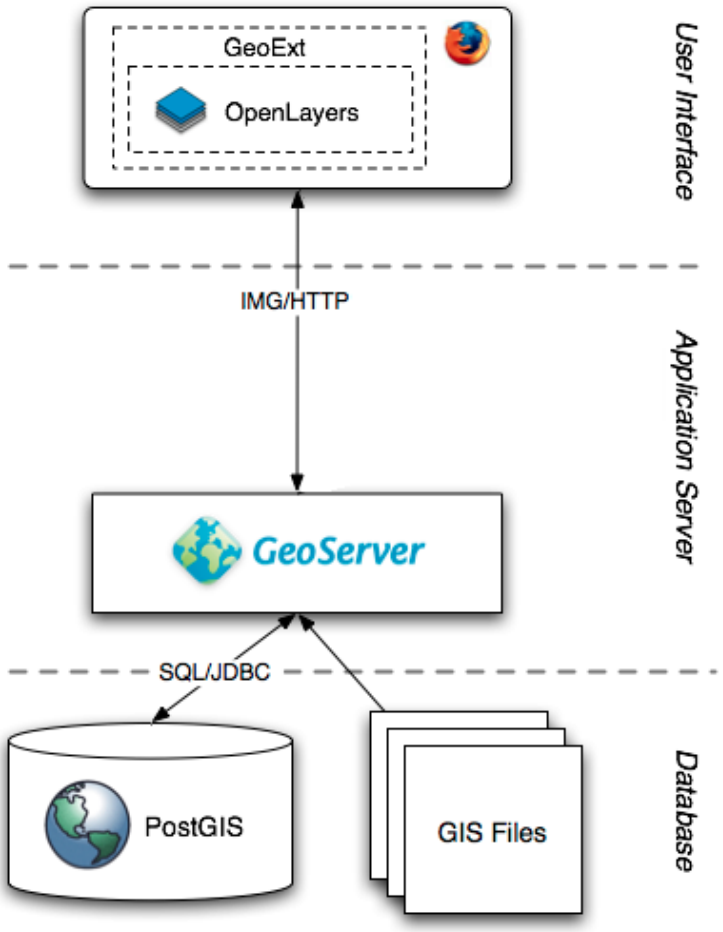


Figure 5.1: Web-Map application structure

5.1.1 OpenLayers

OpenLayers **[13]** is an open source JavaScript library which provides powerful tools for displaying maps in web browsers.

It defines an API for building web-based geographic applications and supports most of the main web-map services such as Google Maps, OpenStreetMap, Yahoo Maps and so on.

As it is mentioned previously, it gives us a great variety of resources when building a web-map service. OpenLayers enables web maps to request map tiles from map servers, create multiple layers on the same map, add features and listeners on it, draw points and routes, among others.

5.1.2 Ext JS

Ext JS **[15]** is a JavaScript library to develop web applications based on Rich Internet Applications RIA [14]. It provides a great amount of widgets to be able to create complex web interfaces.

It enables to create complex grids or panels linked to a layout, a variety of charts, form panels capable of submitting parameters, methods of handling different types of data or data stores which pulls data from web services among others.

5.1.3 GeoExt

GeoExt **[16]** is a JavaScript library which offers the tools for creating rich web mapping applications. It uses the web mapping library OpenLayers and Ext JS for building rich internet applications. GeoExt provides a bunch of widgets to be able to customize widgets and handle geospatial data in order to build applications for viewing, editing and styling geospatial data.

5.2 Deployment Diagram

In this subchapter shall be described application's deployment diagram and it is going to involve what nodes exist (e.g., a web server, an application server or database server), what software components run on each node and finally how they all are connected.

Deployment Diagram is shown below:

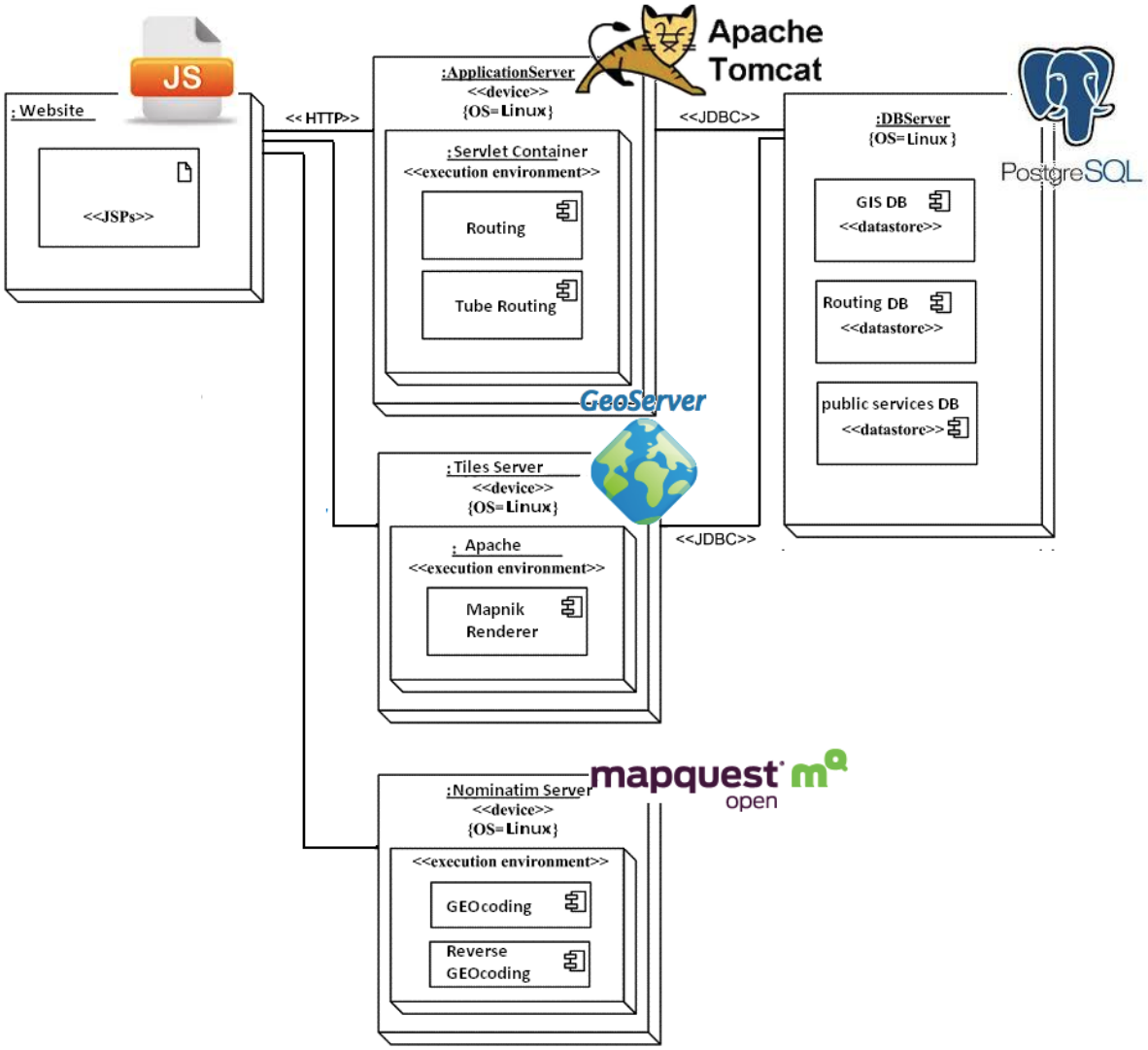


Figure 5.2: Deployment Diagram

In the following list, diagram is analysed on detail:

- *Website* - JavaScript web page where it is designed all the application interfaces which request and shows all the information from servers.

- *Application server* - This server is aimed to process all the routing requests and send back a response with the shortest path. To be able to do it, there is a container of servlets named Tomcat which contains 2 servlets to be able to calculate the shortest path: Routing servlet and Tube Routing servlet. Routing servlet is called every time application needs to know a shortest path between 2 geographical points. When shortest path distance is longer than a certain longitude, Tube Routing servlet is triggered in order to calculate a shortest path taking into account the tube transport.

- *Tiles server* - It runs on Apache server and gives us a tool to provide maps that are sent to OSM web-application. To be able to render maps, it is used Mapnik tool, which makes a rendering process in order to create the map tiles, needed by our web-application. When these tiles are created, they are cached for future requests.

- *Nominatim server* - Nominatim runs as a geocoding converter that is able to return a JavaScript Object Notation JSON file **[17]** with all the possible places or geographical positions from an address or name. It also runs as a reverse geocoding making it possible to get a name or address from its longitude and latitude.

- *DataBase server* - PostgreSQL database that is formed of 3 data stores: gis, routing and public services db. Gis is a database which contains all the OSM data which OpenStreetMap uses to show the maps on the website. Routing database stores all the network data related to ways and nodes in order to make routing queries. Finally, there is a database called public services that provides the tube transport data to be able to route through tube lines.

5.3 Client Side

Client Side subchapter is intended to describe how JavaScript page works, which methods have been used, its structure and how interacts with servers as well. To do this, subchapter is going to be distributed with 2 parts: a workflow and a class diagram.

5.3.1 Workflow Diagram

Each application workflow is going to be explained in the following figure:

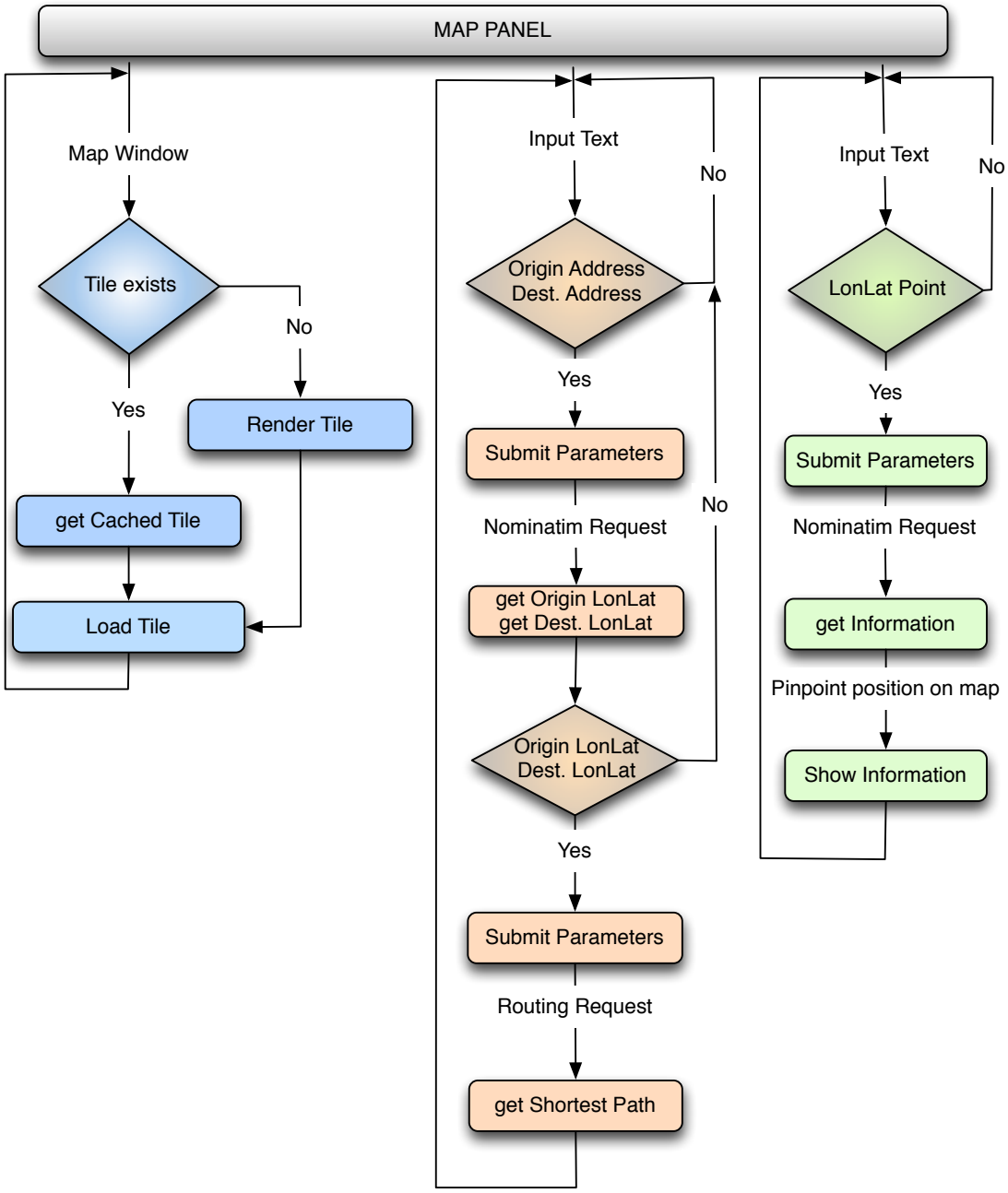


Figure 5.3: Application workflow diagram

- The first workflow highlighted in blue shows us the steps map application must take to render. When slipping the map window, application has to get tile maps from server in order to load them on the map window.

- The second one highlighted in brown, describes the workflow generated when asking for a shortest path routing. First of all, origin and target address points must be filled in to be submitted to Nominatim service. When receiving geographical points from Nominatim, application is ready to submit these points to the Routing service to request the shortest path.

- The last one highlighted in green, is aimed to display how application pinpoints a geographical point and pops up a window displaying all the information about that point. As shows the workflow, first user must fill a geographical coordinates which are going to be submitted to Nominatim to get the entire information about the point. When application gets Nominatim response, it pinpoints a popup window on the map with that information.

5.3.2 Class Diagram

Class Diagram help us structure classes in our application code in order to understand what properties a class is formed of and how classes interact each other.

In the following figure application class diagram is described:

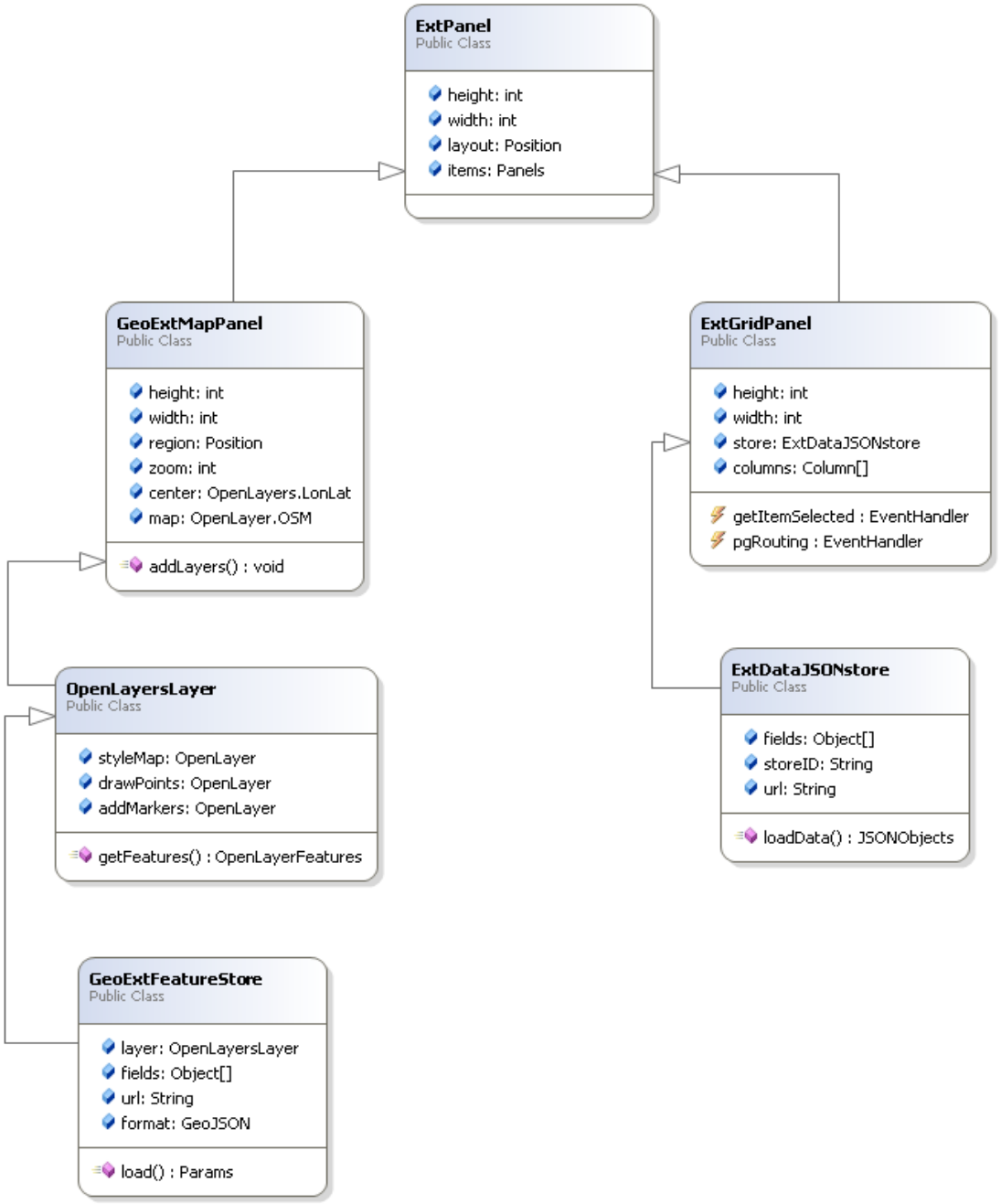


Figure 5.4: Application class diagram

Each class is explained below:

- *ExtPanel* - Class which creates a panel container in order to add different type of panels in our web page.

- *GeoMapPanel* - Class that builds a map panel and integrates OpenLayers class option which is able to implement a OSM map and add different layers on it.

- *OpenLayersLayer* - OpenLayers class to create new layers and add them to OSM map. They are capable of setting a map style on the layer and adding features such as points or ways.

- *GeoExtFeatureStore* - Class to store geographical features such as points or ways. It allows to make requests to our Routing service server in order to get a GeoJSON file containing all the features a shortest path is made of. Then, it loads this information to our map layer.

- *ExtGridPanel* - Panel that is aimed to create a grid in which it loads data into. To do this, it provides a store class which contains all the JSON data that is going to be synchronised with the panel. In addition, it manages listeners in order to get the element of the grid selected and call the Routing function after that.

- ExtDataJSONstore - Store that handles data in JSON format. It is capable of requesting JSON data from a web service server and store it. This class is used to make queries to Nominatim server in order to keep and load that information on the grid panel.

5.4 Server Side

Server Side subchapter is aimed to structure and provide details about how our server works and the steps it follows when it is requested. Subchapter also gives us the representation of all the classes that forms server application and its description as well.

5.4.1 Workflow Diagram

Server workflow is shown and described below:

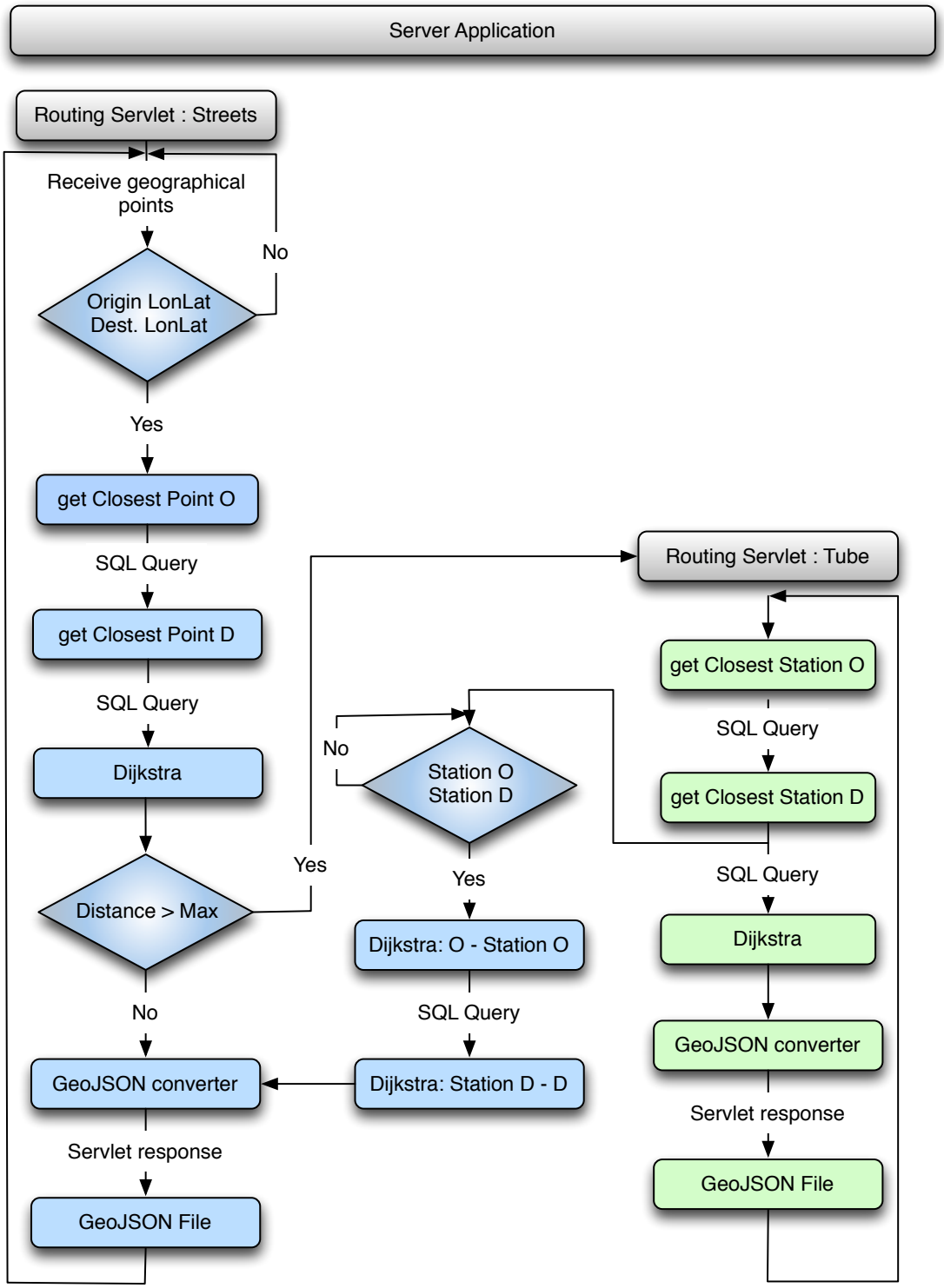


Figure 5.5: Server workflow diagram

- When Streets servlet receives the geographical points (Origin – Destination), it makes an SQL request to database to be able to find the Id of the closest point.

- Once obtained Origin and Destination Ids, servlet is ready to make a routing request to database which is going to return a table with all the ways a shortest path is formed and its distance as well.

- If path distance is quite short, shortest path is ready to be sent to user. Before returning the response, data has to be encapsulated into a GeoJSON file **[18]**. It consists on converting each street of the path as a Java Object with its attributes and transforms them into JSON format. When we got all the information as a JSON format, it is structured as a GeoJSON file. Finally, file is sent to user.

- If path distance is larger, Tube servlet is going to be activated to calculate a path through tube lines. From Origin and Destination geographical points, it makes an SQL request to be able to obtain the closest stations from these points.

- While Tube servlet continues its own process of calculating shortest path through the tube lines, streets servlet must calculate in parallel the shortest path between our origin point and its closest station O as well as the shortest one between our destination point and its closest station D.

- Finally, Streets servlet sends a GeoJSON file containing the shortest path between O – D points and their stations and Tube servlet returns a GeoJSON file with the shortest path through tube lines taking into account these stations.

5.4.2 Class Diagram

Server class diagram is going to describe Routing service structure, the details of each class used and what relation they have between them.

Routing class diagram is described on details below:

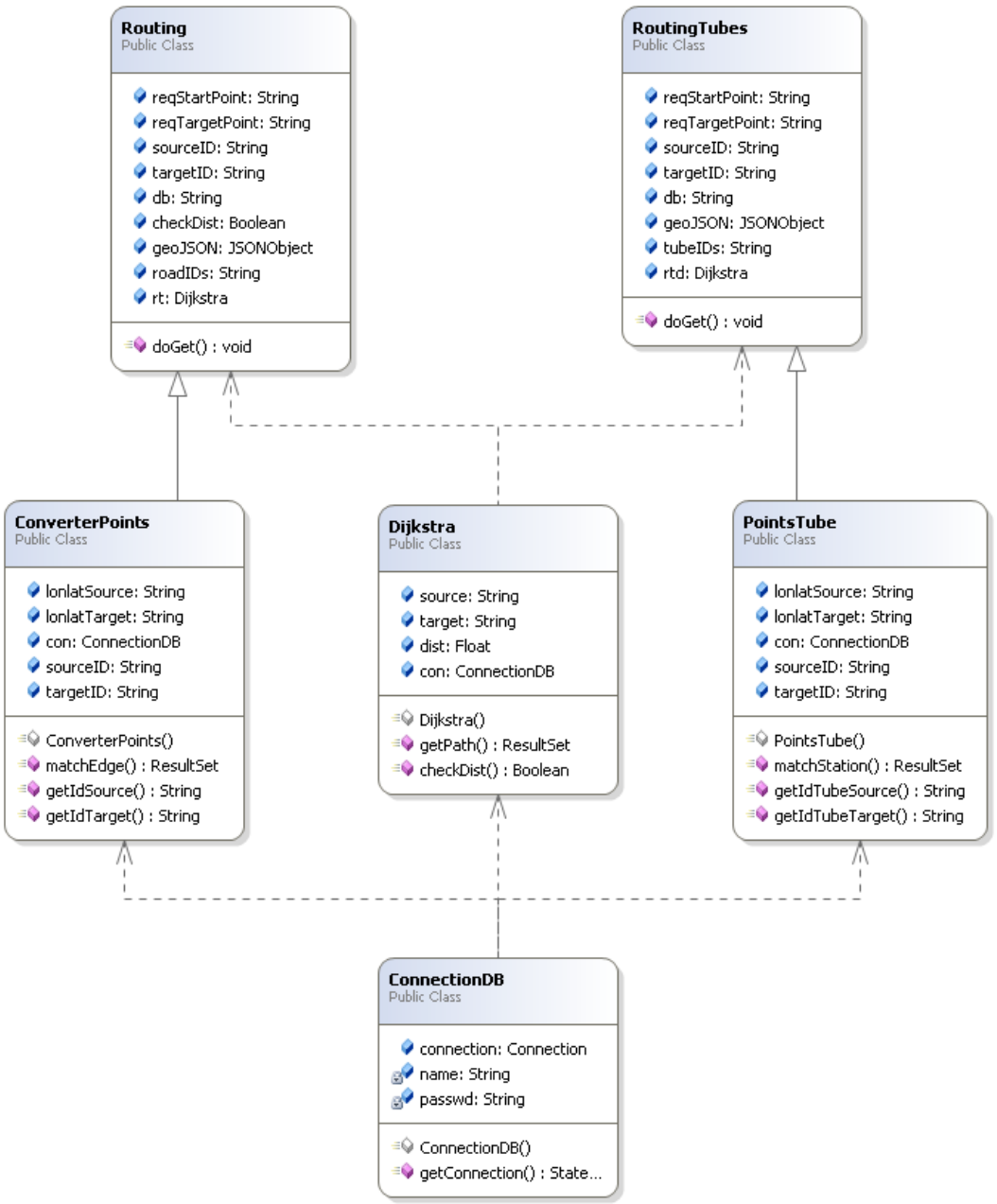


Figure 5.6: Server Class diagram

Where each class has the following target:

- *Routing class* - It implements a servlet class which is able to receive a request and send back a response. In our case, this class receives an origin and destination points and responses a GeoJSON file containing the shortest path information.

- *RoutingTubes class* - In this case, its function is the same as Routing class but it is focused on solving the shortest path taking into account tube lines.

- *ConverterPoints class* - ConverterPoints class goal is to get the closest point IDs from a geographical position by querying it to database. Dijkstra algorithm needs to get point IDs in order to be able to make shortest path queries. It is used by Routing class.

- *PointsTube class* - Its purpose is to get the closest station IDs from a geographical point. It is used by RoutingTube class to be able to request shortest path between tube stations.

- *Dijkstra class* - Dijkstra class provides methods to get the shortest path by making SQL queries to database. Routing database implements Dijkstra algorithm and it provides SQL functions to do the job. This class also implements checkDist method to tell Routing class to route through tube lines.

- *ConnectionDB class* - ConnectionDB builds a class to connect to different databases and implements a method to retrieve this connection by other classes.

5.5 Demo

5.5.1 OSM Web Map Service

OSM Web Map Service demonstration shows us a slippy map window which renders a map while we move into different spots of the map.

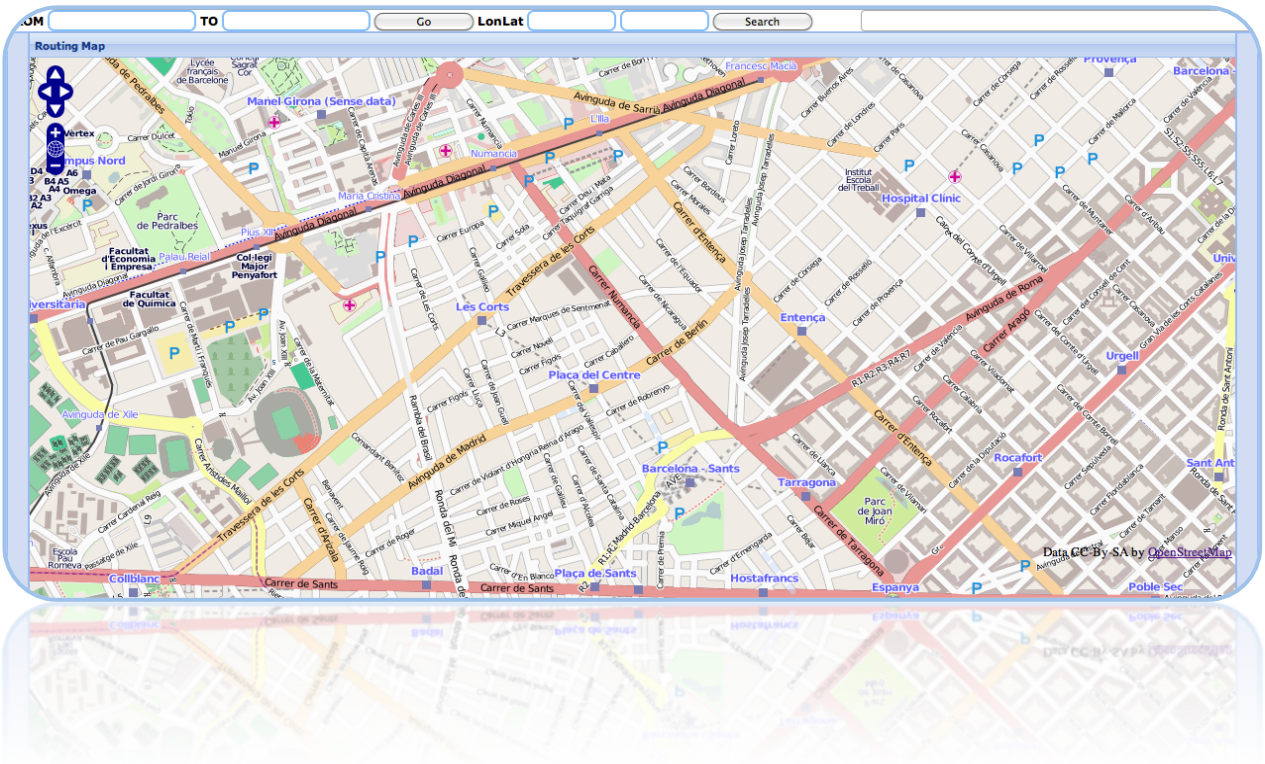


Figure 5.7: OSM Web Map

The figure below shows us all the tiles rendered by our OSM server. Once created, they are sent back to our Web Map application:

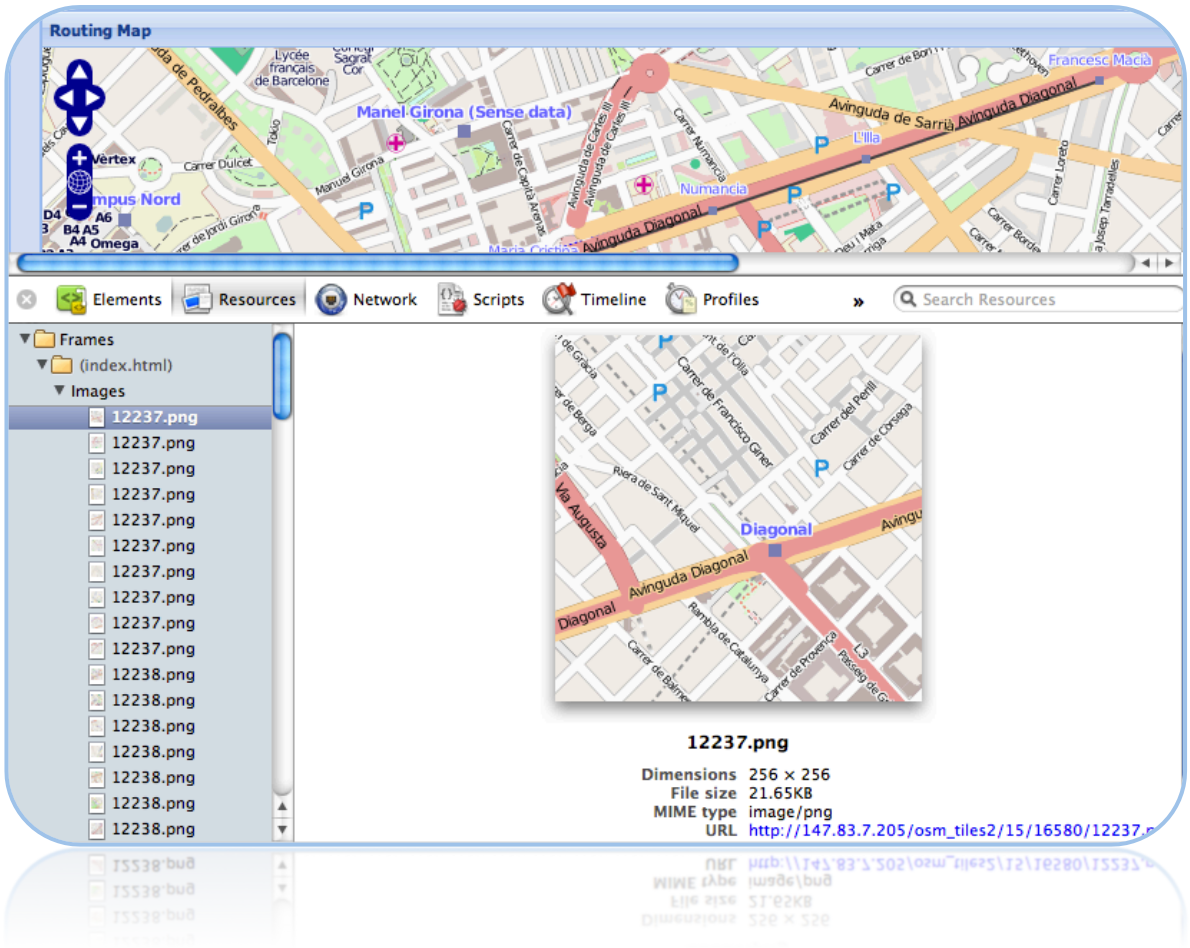


Figure 5.8: Map Tiles requested from application

When requesting maps, Web Map page sends a request to OSM server to get the maps. If the map has not been yet requested, Mapnik tool renders the map tiles and sends them to the user. If the tiles have been cached in previous requests, server just has to provide the cached tiles without having to render them again.

To render tiles, Mapnik is going to connect to GIS database where OSM data has been stored previously.

In the following figure it is shown the steps application must take to render maps:

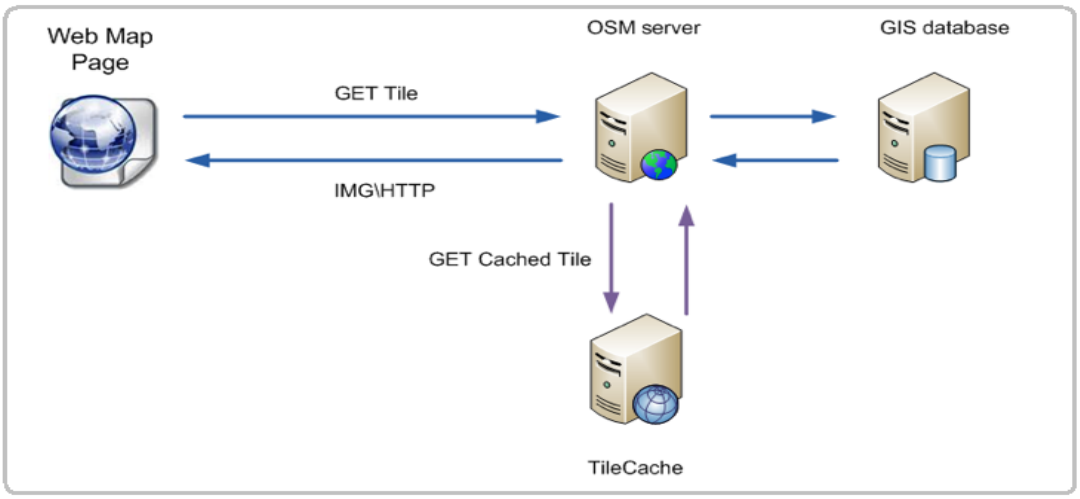


Figure 5.9: Parts involved in rendering maps

5.5.2 Routing Service

To be able to initiate routing service we must fill in this input box with an origin and destination address in order to calculate its shortest path.

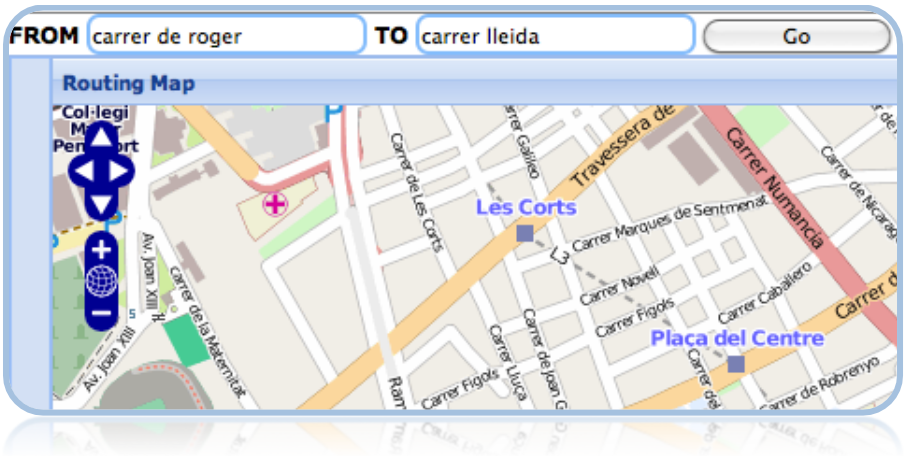


Figure 5.10: Routing text-input box

After submitting the addresses to Nominatim server, it is going to trigger 2 grid panels with all the possible addresses for Origin and Destination points.

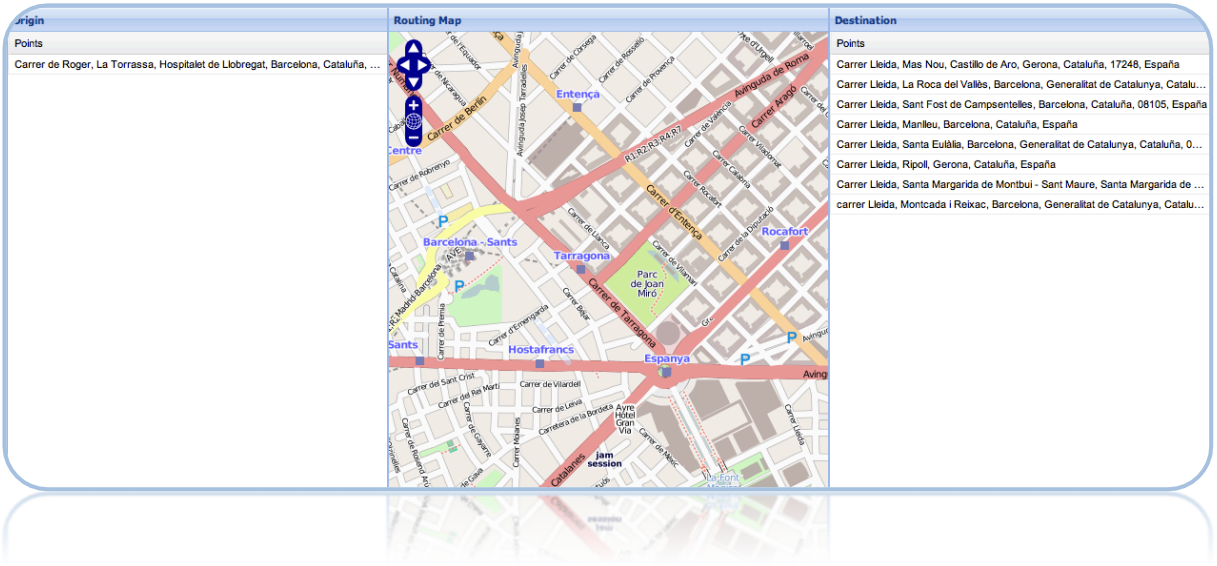


Figure 5.11: Nominatim request

Once clicked Origin and Destination addresses, Nominatim converts addresses to geographical points in order to be able to make the request to Routing server.

Routing server returns a file with the shortest path between these 2 points and they are loaded on the map.

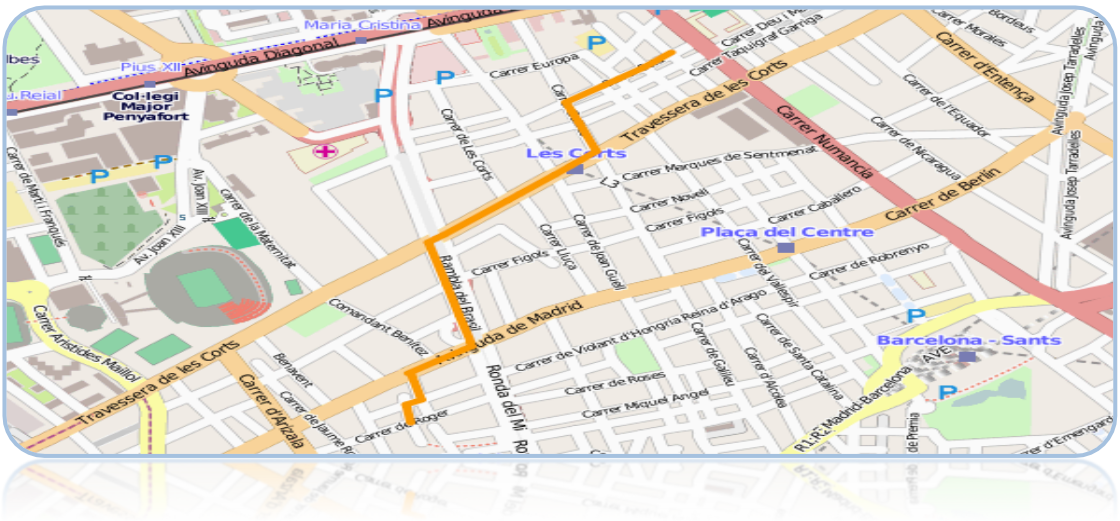


Figure 5.12: Routing response

When the distance between Origin and Destination is large, it is going to take into account Tube transport to get the shortest path.

Figure 5.13: Tube Routing response

To get the geographical points it is needed to make a request to Nominatim request which returns a JSON file containing all the information related to these 2 points.

When Web Map page acquires this information, it extracts Longitude and Latitude coordinates to send them to Routing server. Routing server is going to make shortest path queries to database and it returns a GeoJSON file containing a complete information of the shortest path.

Below is displayed all this process:

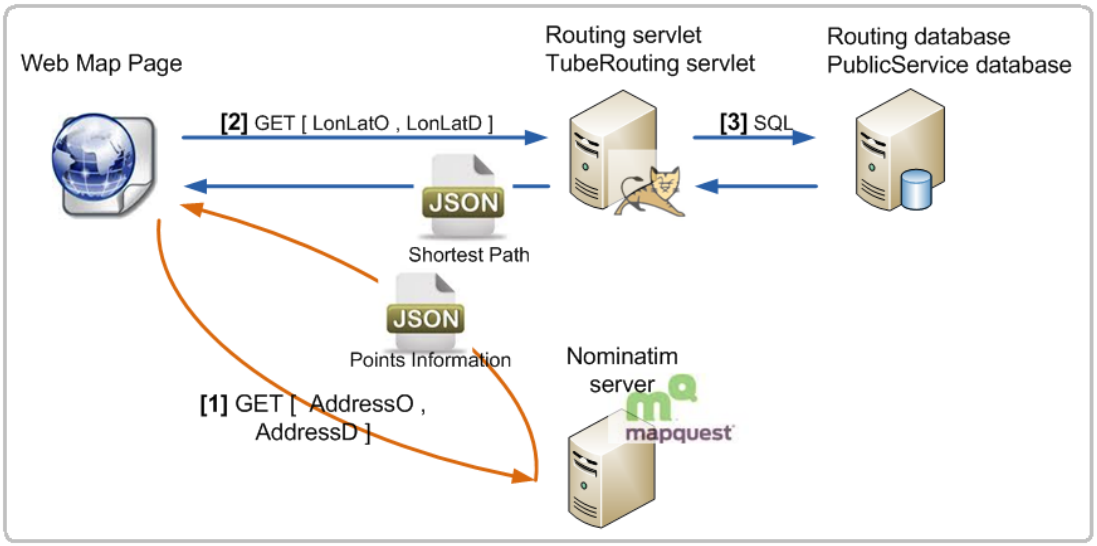


Figure 5.14: Parts involved in requesting for shortest path

5.5.3 Point Information Service

Point Information Service gives us a way to know all the information about a specific point on the map. After filling in a geographical point into search input-text boxes, it will appear a popup window displaying information about the point.

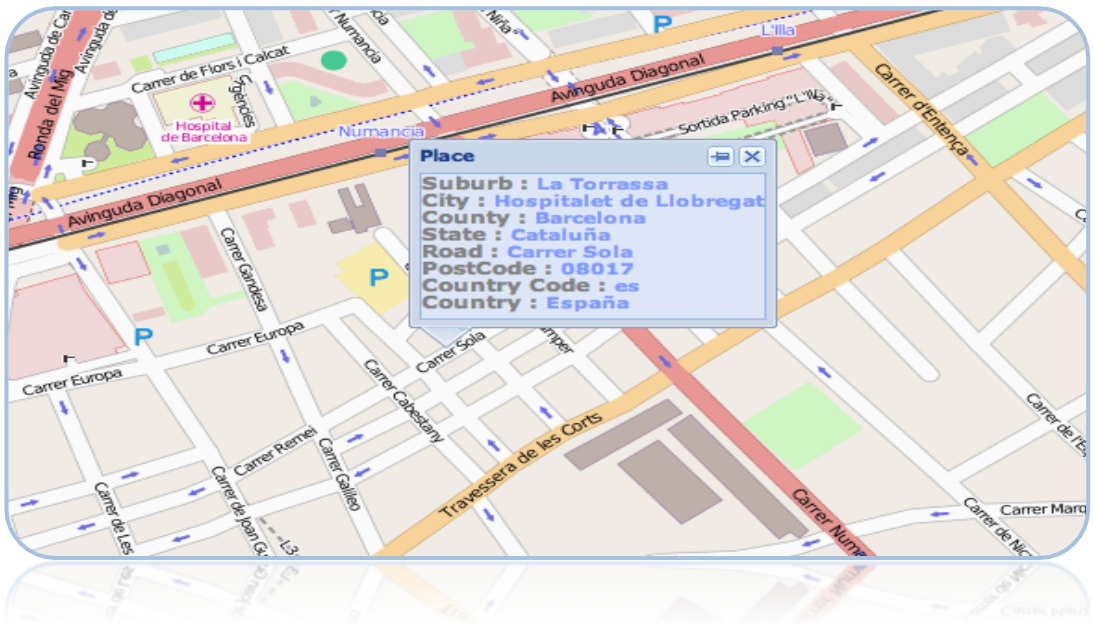


Figure 5.15: Point information window

Once we introduce longitude and latitude points related to a geographical point, web map page sends a request demanding its information. Then, Nominatim send back a JSON file with all the data.

Web Map page extracts a latitude and longitude information from the data obtained to be able to pinpoint a window on the map and load all the data into this window.

Point information process is shown in the following figure:

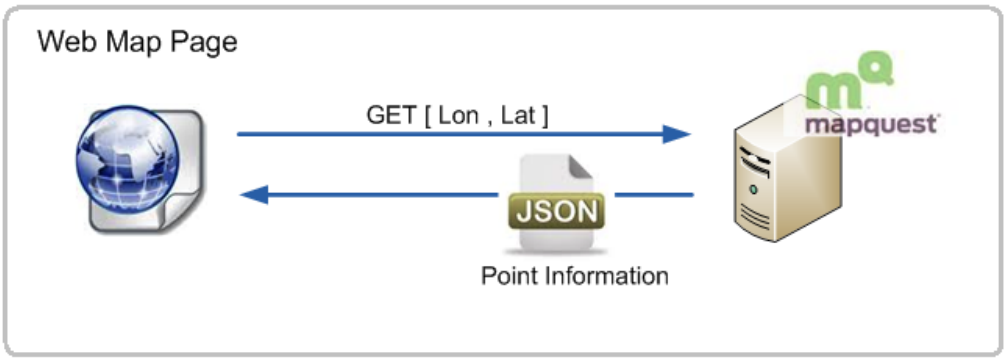


Figure 5.16: Process involved in requesting point information

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

After implementing the entire application we noticed that OSM is capable of offering a wide range of possibilities in terms of customizing maps and features. Therefore, it gives us a total flexibility when it comes to creating your own applications and maps.

In addition, user is able to build up and manage its own OSM server without having to depend on external servers and keep it up-to-date.

Regarding map information, it does not cover as much information as, for instance, Google Maps. In some areas of OSM maps there is a lack of information or at least, some areas are not well detailed. To take an example, OSM offers the option of introducing street numbers but when taking a look in data, there is no such data edited on maps.

Regarding to Nominatim server, it is been impossible for us to build up our own server. Thus, it has been decided to use a public server in order to be able to get a geocoding converter and its performance turned out to be quite good for our purpose.

When decided to implement a Routing service, first Routing goal was to implement and use XAPI server in order to extract the basic OSM information and try to create a street's network to apply Dijkstra algorithm but XAPI server turned out to provide a poor service in terms of response latency.

Taking into account this drawback, we found a project that helped us create a database in which we were able to construct a network from OSM data in order to apply Dijkstra algorithm. Although it has got an awesome result in terms of latency, there is a problem when it comes to its own updating data process, as it does not provide any option of diff-based updating.

6.2 Future Work

It must be said that even though the proposed goals on this project have been achieved, there is a wide margin for improvements. The next steps on this project should be focused on:

- Diff-based updating database - As it is been said in conclusions subchapter, Routing database does not implement tools to update its own data. Thus, it can somehow lead to a problem if Routing database is not updated frequently as it might as well get obsolete. Then, a tool should be implemented for this purpose.

- Routing point granularity - When querying for the shortest path, it does not take into account in which part of a certain street you want to start or finish at. Routing service just assigns you to the closest street without being able to drill down into a more specific point. Therefore, it would be great to achieve, for instance, a routing service where you could indicate house or street numbers as a point to start or finish.

- Quickest path - Shortest path queries do not take into account how much time it needs to get from origin to destination but only its distance. In addition, user does not even know the time scheduler of each transport service and how much time he needs to wait in order to get on a certain transport service. Thus, quickest path should be the fastest way to reach one point to another bearing in mind the time you need to wait for transport service and time to reach destination.

LIST OF FIGURES

LIST OF TABLES

BIBLIOGRAPHY

[1] OpenStreetMap,
“http://en.wikipedia.org/wiki/OpenStreetMap”

[2] OpenStreetMap Elements,
“http://wiki.openstreetmap.org/wiki/Elements”

[3] Map Element Tags,
“http://wiki.openstreetmap.org/wiki/Map_Features”

[4] PostgreSQL,
“http://www.postgresql.org/”

[5] Mapnik,
http://www.mapnik.org/

[6] XAPI,
“http://wiki.openstreetmap.org/wiki/Xapi”

[7] Nominatim,
http://open.mapquestapi.com/nominatim/

[8] JOSM editor,
“http://josm.openstreetmap.de/”

[9] OSM installation,
“http://weait.com/content/build-your-own-openstreetmap-server”

[10] Mod Tile,
“http://wiki.openstreetmap.org/wiki/HowTo_mod_tile”

[11] Nominatim installation,
“http://wiki.openstreetmap.org/wiki/Nominatim/Installation”

[12] pgRouting guide,
“http://workshop.pgrouting.org/chapters/installation.html”

[13] OpenLayers,
“http://docs.openlayers.org/”

[14] Rich Internet Application,
“http://es.wikipedia.org/wiki/Rich_Internet_Applications”

[15] Ext JS,
“http://docs.sencha.com/ext-js/4-0/”

[16] geoExt,
“http://geoext.org/”

[17] JavaScript Object Notation JSON,
“http://www.json.org/”

[18] The GeoJSON format specification,
http://geojson.org/geojson-spec.html

[19] Tomcat Apache
“http://www.h-online.com/open/news/item/Tomcat-7-Apache-Servlet-Container-declared-stable-1169867.html”

[20] Nominatim documentation,
“http://nominatim.openstreetmap.org/”

[21] OpenLayers documentation,
“http://trac.osgeo.org/openlayers/wiki/Documentation”

[22] Routing algorithm documentation,
“http://www.it.uc3m.es/~prometeo/rsc/apuntes/encamina/encamina.html”

[23] Graphs documentation,
“http://www.dma.fi.upm.es/gregorio/grafos/paginagrafos.html”

[24] Dijkstra algorithm documentation,
“http://ict.udlap.mx/people/roberto/dijkstra/index.html”

[25] Min-priority queue,
“http://en.wikipedia.org/wiki/Min-priority_queue”

[26] Min-priority queue,
“http://en.wikipedia.org/wiki/Min-priority_queue”

[27] OSM house numbers,
“http://wiki.openstreetmap.org/wiki/Proposed_features/House_numbers/Karlsruhe_Schema”

[28] How-to create tiles,
“http://wiki.openstreetmap.org/wiki/Creating_your_own_tiles”

[29] Tile cache documentation,
“http://tilecache.org/”

[30] PostgreSLQ documentation,
http://es.wikipedia.org/wiki/PostgreSQL

[31] JQuery JSON requests,
“http://api.jquery.com/jQuery.getJSON/”

[32] How-to cross-domain requests,
“http://aboukone.com/2011/02/04/how-to-post-cross-domain-and-access-the-returned-data-using-extjs/”

[33] JavaScript submit form documentation,
“http://www.thaicreate.com/tutorial/javascript-submit-form.html”

[34] AJAX cross-domain tutorial,
“http://www.bloogie.es/tecnologia/programacion/45-jquery-ajax-desde-diferentes-dominios-cross-domain-ajax”

[35] OSM data sources,
“http://www.geofabrik.de/”

[36] XAPI server implementation,
“https://github.com/iandees/xapi-servlet”

[37] EPSG,
“http://docs.openlayers.org/library/spherical_mercator.html”

ACRONYMS

A

API Application Programming Interface

G

GIS Geographical Information System

GeoJSON Geographical JavaScript Object Notification

J

JSON JavaScript Object Notification

O

OSM OpenStreetMap

P

PostGIS Postgres Geographical Information System

PostgreSQL Postgres Structured Query Language

R

RIA Rich Internet Application

S

SQL Structured Query Language

Servlet Server Applet

X

XAPI OSM Extended API

XML Extensible Markup Language